Laboratorio Nro. 2: Fuerza Bruta

Manuela Valencia Toro
Universidad Eafit
Medellín, Colombia
mvalenciat@eafit.edu.co

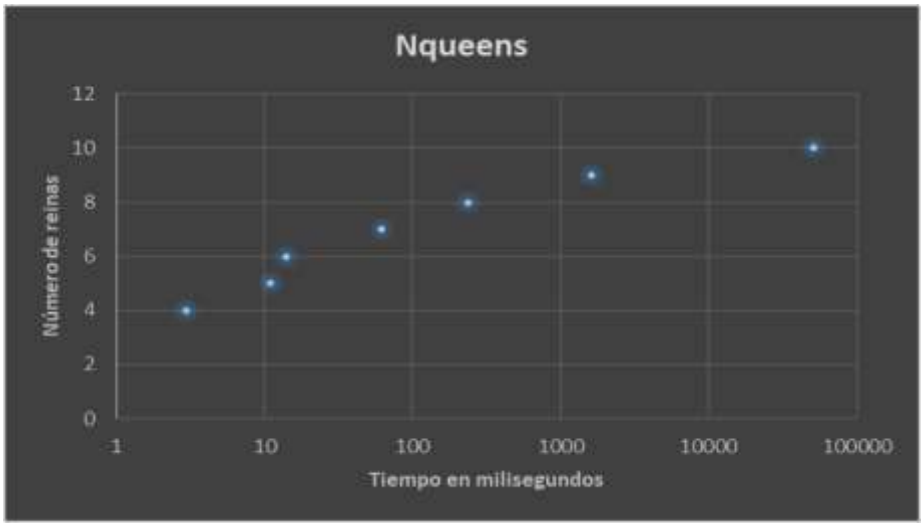
Laura Sánchez Córdoba
Universidad Eafit
Medellín, Colombia
lsanchezc@eafit.edu.co

Felipe Olaya Ospina
Universidad Eafit
Medellín, Colombia
folayao@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

1. Aparte de la fuerza bruta, este problema también puede ser resuelto con backtracking. esta estructura evita estudiar todas las posibles soluciones como lo hace la fuerza bruta, ya que va descartando ciertas combinaciones a medida que avanza si presenta un error. Otra manera de solución puede ser *iterative repair*, la cual consiste en partir de un orden dado y reorganizar a medida que se halle menos cantidad de ataques, sin embargo, esta última puede no obtener resultados.

   Tomado de: https://en.wikipedia.org/wiki/Eight_queens_puzzle#Exercise_in_algorithm_design

2. 

Número de reinas	Tiempo en milisegundos
4	3
5	11
6	14
7	62
8	237
9	1601
10	51312

- Los demás valores no se pudieron tomar ya que a pesar de aumentar el heap en Netbeans, se arrojó OutOfMemoryError

3. El algoritmo parte de la premisa de que para el problema tradicional de las n reinas se debe tener en cuenta que unas casillas del tablero son "prohibidas" por ende se utilizó una matriz que a medida que leyera cada línea guardara la posición de éstas y con el método nReinas que funciona con *backtracking* se verifica que, en primer lugar, la casilla en que se encuentra en programa no sea "un hueco" examinando dicha casilla en la matriz creada y posteriormente se verifica que la reina se pueda poner, Finalmente en un ArrayList se guarda el número de soluciones para cada caso y se imprime.

4. Mediante un scanner se lee cada línea que se introduce por la consola, primero la cantidad de reinas y luego el tablero (cada línea que puede o no tener "un hueco"). Se usó una matriz que tuviese marcados con un -1 los lugares donde no pueden ir las reinas, con la intención de poder tener una representación más sencilla del tablero. Posteriormente se llama al método auxiliar de nReinas que mediante recursión probará en qué casillas puede ir una reina o no bajo dos condiciones: que la posición en la que se piensa poner no sea un "hueco" y, que cumpla con que la reina a poner no ataque las demás (mediante el método puedoPonerReina).

Cabe agregar que el método nReinas funciona con *backtracking*, lo que permite que apenas una solución falle, inmediatamente pase a otra, lo que hace que el algoritmo funcione de manera más eficiente. Finalmente, la respuesta de cada caso de tablero entrado por consola se guarda en un ArrayList para que cuando el programa termine (la entrada sea cero), imprima en orden el número de soluciones posibles para cada caso.

5.

```
//basado en https://github.com/eafit-201710150010/ST0247-
032/blob/master/laboratorios/lab02/ejercicioEnLinea/Lab2.java
import java.util.*;
public class BackQueens

{
   static int hueco [][];
   public static void main(String [] args) {
      ArrayList<Integer> casos = new ArrayList<Integer>(); //c
      String a; //c
      Scanner s = new Scanner (System.in); //c
      int q; //c
      q = s.nextInt(); //c

      while(q!=0){ //n
         hueco= new int [q][q]; //c
         for(int i=0; i<q;i++){ //n
            a=s.next();
            for(int j=0; j<q;j++){ //m

               if (a.charAt(j)=='*'){ //c
                  hueco[i][j]=-1; //c
               }
            }
         }
```

```
        casos.add(nReinas(q)); //O(n)=2n+1
        q=s.nextInt(); //c
        contador=0; //c
    }

    for (int k=0; k<casos.size();k++){ //k
        System.out.println("Case "+(k+1)+": "+casos.get(k));

    }

}
```

T(n)=n((n*m)+2n+1)+k+c
T(n)=n((n*m)+2n+1)+k R.S.
T(n)=n((n*m)+2n+1) R.S.
T(n)=((n*m)+2n+1) R.P.
T(n)=((n*m)+2n) R.S.
T(n)=n*m R.S
O(n)=n*m

```
private static boolean puedoPonerReina(int r, int c, int[] tablero) {
    // complete...
    for (int i = 0; i < r; i++) { //n
        if (tablero[i] == c || (i - r) == (tablero[i] - c) ||(i - r) == (c - tablero[i])) //c
        {
            return false; //c
        }
    }
    return true; //c
}
```

T(n)=n+c
T(n)=n
O(n)=n

```
public static int contador=0;  //c

private static int nReinas(int n) {
    int contador=0; //c
    int[] tablero= new int[n]; //c
    contador=contador + nReinas(0, tablero.length,tablero); //O(n)=2n+1
    return contador;//c
}
```

T(n)=(2n+1)+ c R.S.
T(n)=2n+1
O(n)=2n+1

```
private static int nReinas(int r, int n, int[] tablero) {

    for (int c = 0; c < n; c++) { //n
        if(hueco[r][c]!=-1){ //c
            if (puedoPonerReina(r, c,tablero)) { //n
```

```
            tablero[r] = c; //c
            if (r == n - 1) { //c
               contador++; //c
            } else {
               nReinas(r + 1, n,tablero); //T(n-1)
            }
         }
      }
   }
   return contador; //c

   T(n)=n^2+T(n-1)+c
   T(n)=c+1/6*n*(n+1)(2n+1) //según WolframAlpha
   T(n)=1/6*n*(n+1)(2n+1) R.S.
   T(n)=n*(n+1)(2n+1) R.P.
   T(n)=(n+1)(2n+1) R.P.
   T(n)=2n+1 R.P.
   O(n)=2n+1
}

}
```

6. Las variables n y m hacen referencia al número de filas y columnas de la matriz que representa el tablero de cada uno de los casos ingresados donde están incluidos cada uno de los huecos. Además, se colocó una variable extra k que suma a la complejidad debido a la impresión de las respuestas de los casos (no depende directamente de n o m, solo de la entrada por consola).

4) Simulacro de Parcial

1. **a)** Actual>maximo
   b) O(m*n)

2. **a)** Arr, k + 1
   b) O(n!)

3. **1)** i-m
   2) n
   3) O(m*n)

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co